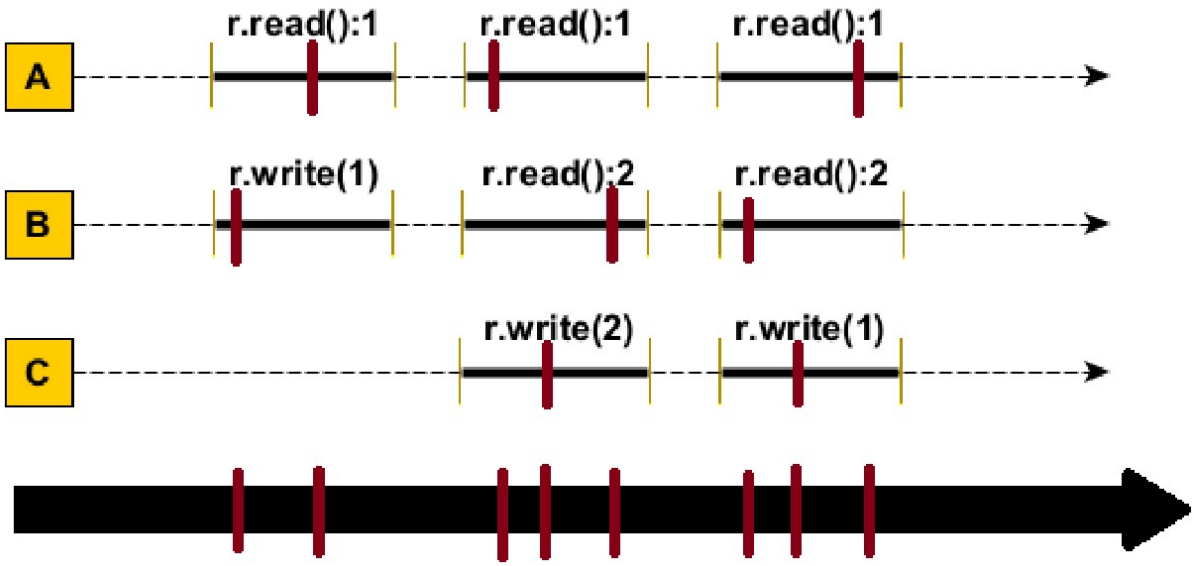
Exercitiul 1

Se da următoarea secvența (istorie) de execuție de mai jos. Este aceasta linearizabila? Dar consistent secventiala? Se considera valoarea initiala r = 0. Argumentați răspunsul oferind explicații (eventual secvența istoriei de execuție) și/sau o diagrama cu punctele de linearizare după caz.



Secvența de execuție este linearizabila fiindcă se poate găsi cel puțin o ordine secvențială ale instrucțiunilor apelate de fiecare thread în acea perioada de timp pentru care toate apelurile sunt satisfăcute din punctul de vedere al corectitudinii.

Secvența de execuție este, de asemenea, și consistent secvențială drept consecință a linearizabilitatii, orice secvență linearizabila este și consistent secvențială.

Exercitiul 2

De ce în mod obișnuit în utilizarea unui lock se prefera ca apelul lock() sa fie executat înainte de blocul try, și nu în cadrul acestuia (prima varianta de mai jos și nu a doua)? Argumentati.

lock inainte de try:

```
someLock.lock();
try {
   .....
}
finally {
   someLock.unlock();
}
```

lock in cadrul try:

```
try {
   someLock.lock();
   .....
}
finally {
   someLock.unlock();
}
```

Apelarea metodei lock este, in mod obisnuit, executată înaintea block-ului try din cauza posibilității intampinarii unei erori de tip unchecked care se poate întâmpla înainte sau în timpul apelului lock în blocul try.

Dacă o astfel de eroare se intampla, firul curent de execuție nu a dobândit acel lock, se va arunca o eroare, se va intra in block-ul finally unde unlock va arunca si el o eroare de tip unchecked fiindcă firul curent nu are acel lock, conform rubricii **Implementation Consideration** ale metodei unlock() din documentatia Java:

“A `Lock` implementation will usually impose restrictions on which thread can release a lock (typically only the holder of the lock can release it) and may throw an (unchecked) exception if the restriction is violated. Any restrictions and the exception type must be documented by that `Lock` implementation.”

În cazul unui lock aflat inafara block-ului try, în cazul intampinarii unei excepții în acel block, metoda unlock din block-ul finally va elibera lock-ul în mod corect datorită dobândirii acelui lock înaintea intrării în try.

Exercitiul 3

De ce in algoritmul Bakery (discutat in cursul 2, al carui pseudocod e amintit mai jos), în comparatia tuplelor din metoda lock (label[i],i) > (label[k],k) nu este utilizata doar comparatia etichetei (label) sau doar a indexului ce identifica threadul, ci e necesara comparatia tuplelor formate din ambele? Argumentați răspunsul descriind o situatie concreta pentru doua thread-uri care ar folosi doar etichetele sau indecsii în comparatia respectivă.

```
class Bakery implements Lock {
  boolean[] flag;
  Label[] label;

  public Bakery (int n) {
        flag  = new boolean[n];
        label = new Label[n];
        for (int i = 0; i < n; i++) {
                flag[i] = false; label[i] = 0;
        }
  }

  public void lock() {
        flag[i]  = true;
        label[i] = max(label[0], ... ,label[n-1])+1;
        while (exists k!=i with flag[k]==true && (label[i],i) > (label[k],k)) {};
  }

  public void unlock() {
        flag[i] = false;
  }
}
```

In algoritmul bakery, rolul etichetei este de a memora o ordine în care firele au intrat în secțiunea critică, fiecare fir are o valoare ce reprezinta a cata intrare in sectiunea critica se realizeaza, iar rolul indexului este de a rezolva potențialele conflicte cauzate de valori identice ale etichetei.

- **Cazul I** (comparatia se realizeaza folosind doar eticheta):
  În cazul în care singurul criteriu de verificare al firelor pentru intrarea în secțiunea citica este valoare etichetei, exista posibilitatea de **blocaj** atunci cand doua fire au aceasi valoare în urma unei atribuiri simultane de eticheta:
    1. T1 intra in lock in acelasi timp ca si T2
    2. T1 si T2 isi seteaza flag la true
    3. T1 și T2 își setează label în același timp (valoarea 3 ca exemplu)
    4. Ambele fire intra in while, T1 este reprezentat de indexul i și T2 este k unde k != i, ambele flag sunt true, deci se trece la comparatie
    5. Algoritmul (din firul T1) așteaptă terminarea execuției tuturor firelor cu eticheta mai mica decat 3, dar îl întâmpina pe T2 cu eticheta cu aceeași valoare, deci continua sa astepte terminarea lui T2 fiindcă label[i] <= label[k]
    6. Algoritmul (din firul T2) intra în aceasi situatie, așteaptă terminarea lui T1 label[i] <= label[k]
    7. Programul ramane suspendat in blocaj

- **Cazul II (**comparatia se realizeaza folosind doar indexul):
  În acest caz pot apărea **probleme de corectitudine** fiindcă programul nu mai garantează respectarea ordinii de intrare în secțiunea critică fiindcă acestea nu mai sunt luate în considerare:
    1. Firul T2 intra in lock, flag[2] = true si label[2] = 1
    2. Firul T1 intra in lock, flag[1] = true si label[1] = 2
    3. Ambele fire intra in while, T2 este reprezentat de indexul i și T1 este k unde k != i, ambele flag sunt true, deci se trece la comparatie
    4. Se compara indecsi (în algoritmul din firul T2), indexul lui T2 este mai mare decat indexul lui T1, deci firul T2 trebuie sa aștepte finalizarea execuției firului T1 în ciuda faptului că valoarea din label ar indica faptul ca ar urma T2 sa intra în secțiunea critică
    5. T1 intra in sectiunea critica, își setează flag-ul pe false si i se permite lui T2 sa intre
    6. În acest moment rezultatul după execuția celor doua fire n-are garanția de a fi corect din cauza nerespectării ordinii de intrare în secțiunea critică

In concluzie, comparatia tuplelor formate din ericheta si index este necesara pentrua prevenii blocaje prin rezolvarea conflictelor de valori identice ale etichetei, și pentru a asigura corectitudinea programului prin urmărirea și respectarea ordinii de intrare ale firelor în secțiunea critică.

4. a) O echipa de programatori a dezvoltat algoritmul de lock prezentat in pseudocodul urmator. *ThreadId* se considera a fi o clasa ce furnizeaza un id unic pozitiv fiecarui thread.
Intr-o executie concurenta a n > 1 thread-uri, este acest algoritm starvation-free? Argumentati.

```
1 class ShadyLock {
2    private volatile int turn;
3    private volatile boolean used = false;
4
5    public void lock() {
6       int me = ThreadId.get();
7       do {
8          do {
9                turn = me;
10         } while (used);
11         used = true;
12      } while (turn != me);
13   }
14
15   public void unlock () {
16      used = false;
17   }
18 }
```

Un algoritm cu proprietatea starvation-free garanteaza ca toate threadurile ce incearca sa acceseze sectiunea critica vor obtine acces intr-un timp finit si vor progresa, fara a risca sa ramana blocate sau sa astepte la nesfarsit din cauza altor threaduri concurente.

In clasa ShadyLock, accesul la linia *turn = me* se realizeaza fara nicio verificare prealabila sau mecanism de prioritizare intre threaduri. Acest lucru poate duce la rescrierea frecventa a variabilei *turn* de catre diferite threaduri, provocand o situatie in care un fir poate fi permanent blocat in bucla *while (turn != me);* . Acest blocaj apare deoarece threadul respectiv nu reuseste sa pastreze valoarea turn pana la accesul efectiv in sectiunea critica, ceea ce face ca algoritmul sa nu fie starvation-free.

Trace demonstrativ:

1. Un fir de executie T1 seteaza *turn = 1* si intra in bucla *while (used);*
2. Un alt fir de executie, T2, seteaza *turn = 2* si intra in bucla *while (used);* inainte ca T1 sa fi setat used = true;
3. Daca used este false si T2 seteaza *used = true* mai rapid decat T1, T2 va reusi sa intre in sectiunea critica, dar T1 va ramane blocat in bucla *while (turn != me)*.

In acest mod, T1 poate ramane in stare de starvation, deoarece nu exista niciun mecanism care sa-i garanteze accesul la sectiunea critica intr-un timp finit.

4. b) O alta echipa de programatori a dezvoltat algoritmul de lock prezentat in pseudocodul urmator ce incapsuleaza un alt lock oarecare. Se considera ca lock-ul incapsulat asigura corect excluderea mutuala si este starvation-free. De asemenea lock-ul incapsulat permite un apel unlock fara exceptie si fara efect chiar daca nu a existat un apel lock. ThreadId se considera a fi o clasa ce furnizeaza un id unic pozitiv fiecarui thread.

Intr-o executie concurenta a n > 1 thread-uri, asigura acest algoritm excluderea mutuala? Argumentati.

```
1 class VeryShadyLock {
2    private Lock lock;
3    private volatile int x, y = 0;
4
5    public void lock() {
6       int me = ThreadId.get();
7       x = me;
8       while (y != 0) {};
9       y = me;
10      if (x != me) {
11         lock.lock();
12      }
13   }
14
15   public void unlock() {
16      y = 0;
17      lock.unlock();
18   }
19 }
```

Excluderea mutuala asigura ca, in orice moment, doar un singur thread poate accesa sectiunea critica.

In clasa VeryShadyLock, variabila volatila x (vizibila tuturor threadurilor) este utilizata pentru a indica threadul care doreste acces la sectiunea critica. De asemenea, variabila volatila y serveste ca indicator al starii sectiunii critice: cand y = 0, sectiunea critica este libera; cand y ≠ 0,

sectiunea critica este ocupata, iar valoarea lui y reprezinta ID-ul threadului care are in acel moment acces la sectiunea critica.

Algoritmul nu asigura excluderea mutuala, deoarece doua thread-uri pot avea acces la sectiunea critica simultan.

Trace demonstrativ:

1. Doua fire de executie T1 si T2 intra in functia lock aproape simultan: T1 seteaza variabila x = 1; si T2 o suprascrie la x = 2.
2. T1 si T2 intra in bucla *while (y != 0) {};* si asteapta eliberarea sectiunii critice.
3. Sectiunea critica se elibereaza si y devine 0, ambele threaduri seteaza *y = me;*
    - T1 seteaza variabila y = 1;
    - T2 o suprascrie la y =2;
4. T1 si T2 verifica *if (x != me)*:
    - T1 trece de aceasta conditie (*if(2 != 1)*) si apeleaza *lock.lock();*
    - T2 nu trece de aceasta conditie (*if(2 != 2)*), deci nu va apela lock-ul si va intra direct in sectiunea critica, simultan cu T1, ceea ce incalca excluderea mutuala.

Exercițiul 4

c.

```
class ShadyChoice {
   private volatile boolean getWhite = false;
   private volatile int last = 0;

   public string choose() {
      int me = ThreadId.get();
      last = me;
      if (getWhite)
         return "white";
      getWhite = true;
      if (last == me)
         return "red";
      else
         return "black";
   }
}
```

Cazul cel mai nefavorabil (Worst Case): Presupunem că accesarea funcției `choose` se face simultan de către n thread-uri. Definim următoarele evenimente:

- Evenimentul e_0: setarea `last = me` la timpul t_0
- Evenimentul e_1: verificarea `getWhite == true` la timpul t_1
- Evenimentul e_2: setarea `getWhite = true` la timpul t_2
- Evenimentul e_3: verificarea `last == me` la timpul t_3

Unde timpii t_0, t_1, t_2 și t_3 sunt aceiași pentru toate cele n thread-uri, iar $t_0 < t_1 < t_2 < t_3$, conform ordinii de execuție. Întrucât accesul este simultan, rezultă următoarele:

- Conform relației $t_1 < t_2$, niciun thread nu va obține culoarea “white”.
- Conform relației $t_0 < t_3$, fiecare thread rescrie imediat valoarea precedentă, valoarea finală în `last` fiind cea a unui thread i. Astfel, pentru n - 1 thread-uri, condiția `last == me` va fi falsă, ceea ce înseamnă că acestea vor obține culoarea “black”, în timp ce thread-ul i va primi culoarea “red”.

Cazul mediu (Average Case): În orice situație în care există o deviație de timp între thread-uri, va exista un thread T_i (unde $1 \leq i < n$) al cărui timp t_2 va fi mai mic decât t_1 pentru n - i thread-uri (care, în consecință, vor primi culoarea “white”). Cele i thread-uri simultane cu T_i vor avea id-ul suprascris, ultima valoare a variabilei `last` fiind:

- i, dacă t_3 (al thread-ului i) < t_0 (pentru orice thread j ∈ {i+1, n}), caz în care T_i primește culoarea “red”
- sau j pentru orice j ∈ {i+1, n} în caz contrar, astfel încât i thread-uri primesc culoarea “black”.

În concluzie, cel mult un singur thread va primi culoarea “red” și cel mult n - 1 vor primi culoarea “black”.

Exercițiul 5

Îmbunătățiri:

1. Adaugarea unui contor de acces pentru fiecare fir de execuție:
   ```
   private static AtomicIntegerArray accessCount = new AtomicIntegerArray(n);
   ```
2. Verificarea contorului înaintea avansarii la un nou nivel:

   ```
   if (k != i && level.get(k) >= L && (victim.get(L) == i || accessCount.get(i) > accessCount.get(k)))
   ```

   În plus față de condițiile inițiale, condiția “sau” dintre contorul de acces și victima curentă este folosită pentru a preveni situațiile în care trei sau mai multe thread-uri așteaptă să avanseze la același nivel, iar un thread mai recent (dar nu cel desemnat drept victimă) avansează înaintea primului thread care așteaptă, fără a se lua în considerare cât de frecvent a accesat acesta secțiunea critică. Această condiție asigură respectarea principiului de fairness, chiar și atunci când mai multe thread-uri concurează pentru a avansa la același nivel.

Fairness integrat – În acest mod, prin rotația bazată pe `accessCount`, implementăm fairness fără o metodă externă de verificare. Algoritmul permite accesul echitabil, deoarece fiecare thread trebuie să aștepte până când toate celelalte au accesat secțiunea critică de un număr aproximativ egal de ori.

Astfel, această modificare asigură 0-bounded waiting și fair access la secțiunea critică.